

# Running PEs Inline Without a Console

---

 [coresecurity.com/core-labs/articles/running-pes-inline-without-console](https://coresecurity.com/core-labs/articles/running-pes-inline-without-console)

While reading the amazing [Inline-Execute-PE](#) by [Octoberfest7](#), I noticed that to obtain the output from the PE being executed, the author needed to allocate a console, which results in a process being created (conhost.exe).

Interestingly, the readme also states that a commercial C2 managed to avoid spawning a conhost.exe process by "fooling Windows into thinking it had a console." After reading this, I thought I might give it a try and attempt to achieve the same.

This blogpost is the result of that research project, which took me three weeks of demanding work and led to some interesting results.

## A Brief Introduction to How Consoles Work

---

To put it simply, a console is the black box you see when you run CMD. Programs can get user input from it and print output to it.

In Windows, the console is run by a separate process called conhost.exe, which interacts with the actual executable via a series of APIs. But not all processes have a console, some have a GUI like notepad or run in detached mode like lsass. If a process wants to allocate a console, all it must do is call [AllocConsole](#), which will create the conhost.exe process, initialize the standard input, output and error streams, and save a handle to the console under `PEB->ProcessParameters->ConsoleHandle`.

Given that different binaries interact with the console in diverse ways, convincing binaries that we already have a console (when we do not) and redirecting its output to a pipe will require not just one trick, but several. I will go over the techniques that were used for each binary going from the simplest to the most complex.

## Hello World with MinGW

---

I started with a simple "hello world" project in C which I cross-compiled from Linux to Windows with the MinGW compiler.

Before loading and running this PE inline (meaning, in the same process as Beacon), I allocated a console with [AllocConsole](#). This created a conhost.exe process; initialized the standard input, output, and error streams; and set the new `ConsoleHandle` on the PEB. Also, I updated the standard output and error streams by calling [SetStdHandle](#) with the write handle of the anonymous pipe I created previously.

```
AllocConsole();
SetStdHandle(STD_OUTPUT_HANDLE, hPipeWrite);
SetStdHandle(STD_ERROR_HANDLE, hPipeWrite);
```

Next, I invalidated the *ConsoleHandle* on the PEB so that the program could not interact with the console anymore.

```
PEB->ProcessParameters->ConsoleHandle = 0x123;
```

After running the program, I noticed that the redirection still worked, which seemed to indicate that the program did not actually interact with the console directly.

Knowing this, I decided to modify the *stdout* FILE\* structure in memory, which is defined below:

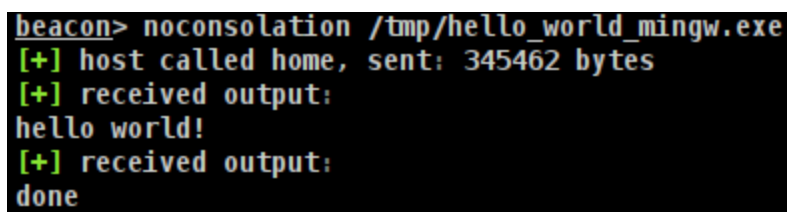
```
struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char *_tmpfname;
};
typedef struct _iobuf FILE;
```

The *\_file* attribute is supposed to be the file descriptor, but because we are using an anonymous pipe, we only have a handle. To convert the *pipe* write handle to a file descriptor I called *\_\_open\_osfhandle* and used the resulting descriptor to set *stdout->\_file*.

The last change needed was to set the flags (*\_flag*) to *\_IOWRT* (file descriptor is writable) and *\_IONBF* (disable buffering).

After this, I successfully redirected the output without having to allocate a console.

Image



```
beacon> noconsolation /tmp/hello_world_mingw.exe
[+] host called home, sent: 345462 bytes
[+] received output:
hello world!
[+] received output:
done
```

## Hello World with MSVC

---

When compiling the “Hello World” program with the Microsoft compiler, I tried the same trick from before and it seemed to work fine, but when I tried to run a more complicated program (nanodump compiled with the MSVC compiler) I noticed that I got no output.

It appears that if you compile your program like this, it works:

```
cl.exe helloworld.c /Fe:helloworld.exe
```

However, if you compiled like this, it does not:

```
cl.exe helloworld.c -c -nologo
link.exe /OUT:helloworld.exe -nologo libvcruntime.lib libcmtd.lib ucrt.lib
kernel32.lib /MACHINE:X64 -subsystem:console -nodefaultlib helloworld.obj
```

When compiled like the second instance, the binary uses the new C Runtime implementation from *ucrtbase* instead of the legacy *msvcrt*. While investigating the issue in [Ghidra](#), I realized that the definition of the function *fileno* is different between *msvcrt.dll* and *ucrtbase.dll*.

```
msvcrt!fileno
...
mov     eax, dword ptr [rcx+1Ch]
add     rsp, 38h
ret
```

```
ucrtbase!_fileno:
...
mov     eax, dword ptr [rcx+18h]
add     rsp, 28h
ret
```

The offset of the *\_file* attribute inside the FILE structure in *msvcrt.dll* is 0x1C and in *ucrtbase.dll* it is 0x18, which means that the FILE structures are not the same. This explains why the previous approach was not working, I was writing the *\_file* and *\_flag* values at the wrong offsets.

I reverse engineered the definition of the FILE structure as it is used in *ucrtbase.dll*:

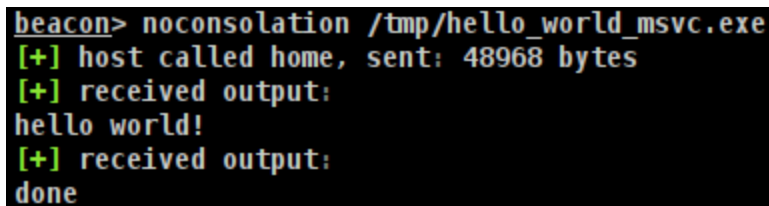
```
typedef struct _UCRTBASE_FILE {
/*0x00 0x08*/ PVOID  _ptr;
/*0x08 0x08*/ PVOID  _base;
/*0x10 0x04*/ UINT32 _cnt;
/*0x14 0x04*/ UINT32 _flags;
/*0x18 0x04*/ UINT32 _file;
/*0x1c 0x04*/ UINT32 _bufsiz;
/*0x20 0x08*/ PVOID  _charbuf;
/*0x28 0x08*/ LPSTR  _tmpfname;
/*0x30 0x28*/ CRITICAL_SECTION _lock;
} UCRTBASE_FILE, * PUCRTBASE_FILE;
```

Just updating the FILE definition is not enough though, because the flags changed as well. After some debugging, I determined that the flags need to be 0x2402. Also, as per Window's [documentation](#), we need to set the *LockCount* attribute on the *\_lock structure* to – 1.

Finally, I realized that I needed to call the function *\_open\_osfhandle*, which is defined at *ucrtbase.dll* instead of the old *msvcrt.dll*, for the redirection to work.

With all these changes, I managed to reproduce the previous technique to a binary compiled with MSVC.

Image



```
beacon> noconsolation /tmp/hello_world_msvc.exe
[+] host called home, sent: 48968 bytes
[+] received output:
hello world!
[+] received output:
done
```

## CMD

---

An important binary that I wanted to support is *cmd.exe*, which has its own set of challenges, because when one runs *cmd.exe /c whoami*, this is not the *cmd.exe* that resolves who the current user is, but the *whoami.exe* binary which is located at *C:\Windows\System32\whoami.exe*. So, we want to be able to obtain the output of the processes that *cmd.exe* creates.

If we allocate a console and set the write handle of the pipe as the StandardOutput and StandardError, we actually do get the output of the *whoami.exe* process.

```
AllocConsole();
SetStdHandle(STD_OUTPUT_HANDLE, hPipeWrite);
SetStdHandle(STD_ERROR_HANDLE, hPipeWrite);
```

This means that *cmd.exe* is indeed capable of communicating both the current console and our desired output handles to its child processes. However, what we want is *cmd.exe* to pass over our output handles without needing to allocate a console. To do this, we need to set the *ConsoleHandle* on the child processes to  $-1$ , indicating that there is no console allocated, only the output handles.

To prepare the parameters that will be passed onto the new process, *CreateProcessW* will call *BasepCreateProcessParameters*. This function will read some values from an undocumented internal structure called *ConsoleConnectionState*, which is populated by *AllocConsole*. I have reverse engineered its fields:

```
typedef struct _CONSOLE_CONNECTION_STATE {
/*0x00 0x01*/ BYTE   Flags;
/*0x08 0x08*/ HANDLE ConsoleHandle;
/*0x10 0x08*/ HANDLE ConsoleReference;
/*0x18 0x08*/ HANDLE StandardInput;
/*0x20 0x08*/ HANDLE StandardOutput;
/*0x28 0x08*/ HANDLE StandardError;
/*0x30 0x01*/ BYTE   IsConnected;
} CONSOLE_CONNECTION_STATE, * PCONSOLE_CONNECTION_STATE;
```

Now, let's see an illustrative code snippet from *BasepCreateProcessParameters* where the *ConsoleHandle* for the child process is set:

```
ChildProcParams->ConsoleHandle = ConsoleConnectionState.ConsoleReference;
if (ChildProcParams->ConsoleHandle == NULL) {
    ChildProcParams->ConsoleHandle = PEB->ProcessParameters->ConsoleHandle;
}
```

The *ConsoleHandle* is set to *ConsoleConnectionState.ConsoleReference* and if that is *NULL*, then it is set to the *ConsoleHandle* of the current process.

Meaning, we want the *ConsoleReference* to be *NULL* and our own *ConsoleHandle* to be  $-1$ . To put it even more simply, set the *ConsoleReference* to  $-1$ . The question is, how can we get the base address of this internal structure?

To find it, I relied on a function called *BaseGetConsoleReference*, which returns the *ConsoleReference*, like so:

```
HANDLE BaseGetConsoleReference(void)
{
    return ConsoleConnectionState.ConsoleReference;
}
```

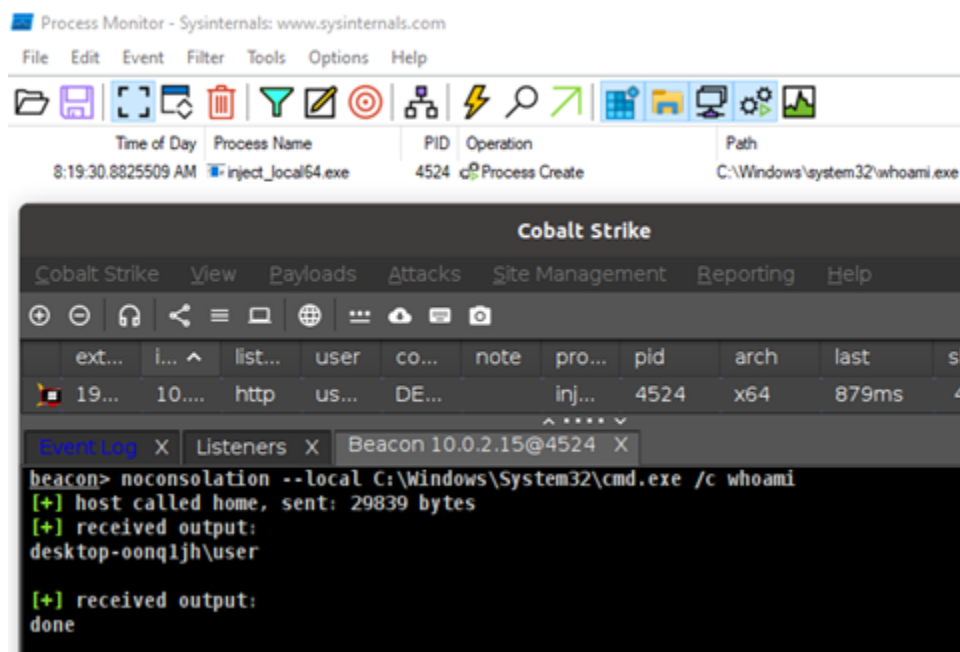
But we do not care about the value of the `ConsoleReference`, we care about the address of the `ConsoleConnectionState` structure. So, I simply parsed the assembly of this function to find it.

```
48 8b 05 f9 94 19 00    mov     rax,QWORD PTR [rip+0x1994f9]
c3                               ret
```

To obtain its address, we need to extract the offset used by the `mov` instruction, which in this case is `0x1994f9`. Then we add the address of the `ret` instruction (this is because the function uses RIP-relative addressing). The result will be the address of the `ConsoleReference`, so to get the base of the entire structure, we just subtract the offset of the field, which is `0x10` bytes.

Once we know where this structure is, we set the `ConsoleReference` to `-1`, set the `StandardOutput` and `StandardError` to the write handle of our pipe and we can now get the output of the commands we run via `cmd.exe`.

Image



## PowerShell

This project would not be complete without the ability to run PowerShell without a console.

This is by far the most complicated piece of the puzzle, as the PowerShell process is intimately related to the console and decoupling them is no easy task.

First, I allocated a console, and invalidated the `ConsoleHandle`.

```
AllocConsole();
SetStdHandle(STD_OUTPUT_HANDLE, hWrite);
SetStdHandle(STD_ERROR_HANDLE, hWrite);
PEB->ProcessParameters->ConsoleHandle = 0x123;
```

After doing this, I got no output, which means that the PowerShell process truly needs the console to be valid, or we get no output.

To understand exactly where the handle is used, I configured a hardware breakpoint on WinDbg that will break upon read access to the *ConsoleHandle*. The *ConsoleHandle* is stored in the ProcessParameters structure, which is referenced by the PEB, so I got its address (which in my case, was 0xF1F40) and configured the hardware breakpoint, like so:

```
0:005> ba r 8 0xF1F40 "k;g"
```

Every time that a function reads the console handle, the stack trace is going to be printed on the screen and then it is going to continue executing.

After a few seconds, I had the full list of functions that read the *ConsoleHandle*. After some cleaning up, I got the following list:

- SetThreadUILanguage
- SetThreadPreferredUILanguages2
- GetConsoleCP
- GetCurrentConsoleFontEx
- GetConsoleMode
- GetConsoleScreenBufferInfo
- GetConsoleScreenBufferInfo
- GetConsoleMode
- SetConsoleMode
- GetConsoleMode
- GetConsoleMode
- GetConsoleMode

Once I knew which functions used the *ConsoleHandle*, I patched them in memory and replaced them with my own dummy implementation, that did nothing and returned successfully.

To be sure my modifications did not break the inner workings of PowerShell, I ran the following test:

```
AllocConsole();
SetStdHandle(STD_OUTPUT_HANDLE, hWrite);
SetStdHandle(STD_ERROR_HANDLE, hWrite);
patchKernelbase();
```

After a few bug fixes, I managed to obtain the PowerShell output, which meant my dummy implementations were working properly. I then re-ran the previous test, but this time, I invalidated the *ConsoleHandle*:

```
AllocConsole();
SetStdHandle(STD_OUTPUT_HANDLE, hWrite);
SetStdHandle(STD_ERROR_HANDLE, hWrite);
patchKernelbase();
PEB->ProcessParameters->ConsoleHandle = 0x123;
```

I was confident this was going to work, but to my surprise, it did not.

I thought I was surely missing some API, so I re-ran the test with the previous hardware breakpoint and got zero hits, which meant no API was reading the *ConsoleHandle*. How can it be that modifying a memory address that no one reads breaks the output redirection? I figured WinDbg was missing a read somehow and decided to continue testing.

Instead of invalidating the *ConsoleHandle* before running PowerShell, I decided to do it inside one of my dummy functions, which meant that the handle would be invalidated during the execution of PowerShell and not before.

I tried this on all the dummy functions, one by one, and realized that some functions allowed me to invalidate the handle (meaning I managed to redirect the output) and some did not.

After some cleanup, I ended up with the following list:

- X SetThreadUILanguage
- X SetThreadPreferredUILanguages2
- X GetConsoleCP
- ✓ GetCurrentConsoleFontEx
- ✓ GetConsoleMode
- ✓ GetConsoleScreenBufferInfo
- ✓ GetConsoleScreenBufferInfo
- ✓ GetConsoleMode
- ✓ SetConsoleMode
- ✓ GetConsoleTitleW
- ✓ GetConsoleMode
- ✓ GetConsoleTitleW
- ✓ GetConsoleMode
- ✓ SetTEBLangID
- ✓ SetConsoleTitleW
- ✓ GetConsoleMode
- ✓ SetThreadUILanguage
- ✓ GetConsoleOutputCP



- ✓ GetConsoleScreenBufferInfo
- ✓ GetConsoleOutputCP
- ✓ SetThreadUILanguage
- ✓ GetConsoleOutputC

From the output above, it was evident that something had to be happening in between *GetConsoleCP* and *GetCurrentConsoleFontEx*. And if we analyze the stack trace for both these function calls, we learn that they are called by the same function:

```
# Child-SP RetAddr Call Site
00 KERNELBASE!GetConsoleCP
01 Microsoft_PowerShell_ConsoleHost_ni+0x71563
02 Microsoft_PowerShell_ConsoleHost_ni!Microsoft.PowerShell.ConsoleControl.UpdateLocaleSpecificFont+0x24
...
```

```
# Child-SP RetAddr Call Site
00 KERNELBASE!GetCurrentConsoleFontEx
01 Microsoft_PowerShell_ConsoleHost_ni+0x73912
02
Microsoft_PowerShell_ConsoleHost_ni!Microsoft.PowerShell.ConsoleControl.GetConsoleFontInfo+0x78
03
Microsoft_PowerShell_ConsoleHost_ni!Microsoft.PowerShell.ConsoleControl.UpdateLocaleSpecificFont+0x5a
...
```

Interestingly, the code for *UpdateLocaleSpecificFont* is public and can be found [here](#). The relevant code snippet is:

Image

```

internal static void UpdateLocaleSpecificFont()
{
    // Default Powershell shortcut.lnk settings.
    const string defaultFontFace = "Lucida Console";

    // Default CJK locale shortcut.lnk settings.
    // Font size is hard coded here to ensure we select a supported size.
    // GDI does a poor job of selecting raster font size if the requested
    // size is not supported.
    const string CJKFontFace = "Terminal";
    const int CJKFontFamily = 48;
    const int CJKnFont = 6;
    const int CJKFontWidth = 8;
    const int CJKFontHeight = 12;
    const int CKJFontWeight = 400;

    uint currentLocaleCodePage = (uint)ConsoleControl.NativeMethods.GetConsoleCP();

    ConsoleHandle handle = ConsoleControl.GetActiveScreenBufferHandle();
    CONSOLE_FONT_INFO_EX fontInfo;
    try
    {
        fontInfo = ConsoleControl.GetConsoleFontInfo(handle);
    }
    catch (Exception)
    {
        return;
    }
}

```

At the start, we can see the call to *GetConsoleCP*, and at the end, the call to *GetCurrentConsoleFontEx*. This means that the issue lies in whatever *GetActiveScreenBufferHandle* is doing.

After we inspect its code, we learn that it calls this function right [here](#):

Image

```

static readonly Lazy<ConsoleHandle> _outputHandle = new Lazy<SafeFileHandle>(() =>
{
    // We use CreateFile here instead of GetStdWin32Handle, as GetStdWin32Handle will return
    // redirected handles
    var handle = NativeMethods.CreateFile(
        "CONOUT$",
        (UInt32)(NativeMethods.AccessQualifiers.GenericRead | NativeMethods.AccessQualifiers.
            GenericWrite),
        (UInt32)NativeMethods.ShareModes.ShareWrite,
        (IntPtr)0,
        (UInt32)NativeMethods.CreationDisposition.OpenExisting,
        0,
        (IntPtr)0);
    if (handle == NativeMethods.INVALID_HANDLE_VALUE)
    {
        int err = Marshal.GetLastWin32Error();
        HostException e = CreateHostException(err, "RetrieveActiveScreenBufferConsoleHandle",
            ErrorCategory.ResourceUnavailable, "The Win32 internal error \"{0}\" 0x{1:X}
            occurred while retrieving the handle for the active console output buffer.
            Contact Microsoft Customer Support Services." );
        throw e;
    }
    return new ConsoleHandle(handle, true);
});

```

The documentation from [CreateFile](#) reads:

*CONOUT\$* gets a handle to the active screen buffer, even if *SetStdHandle* redirects the standard output handle.

This actually makes a lot of sense. PowerShell is calling *CreateFile* with *CONOUT\$*, which will end up calling *NtCreateFile*, which will be handled by the Windows kernel. Apparently, the kernel reads the *ConsoleHandle* of the calling process to service this call, which

explains why the hardware breakpoint was not being triggered—it was being read from kernel-land.

If the *ConsoleHandle* is invalid (which it is in our case), *CreateFile* fails and returns `INVALID_HANDLE` which will make *GetActiveScreenBufferHandle* throw a “HostException” which *UpdateLocaleSpecificFont* will not catch. This means that the call to *CreateFile* has to succeed if we want to be able to redirect PowerShell’s output.

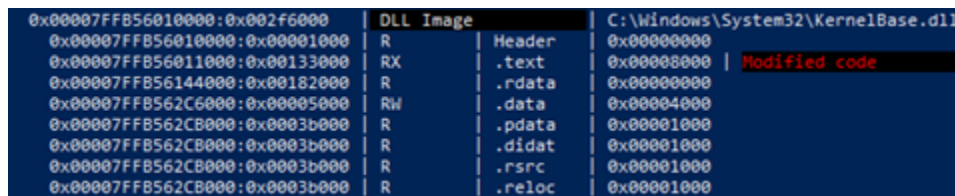
Now that we know which functions are important, let’s discuss our options of how we can redirect the output of PowerShell without allocating a console.

## Memory Patching

---

We already know that directly modifying the instructions from the functions that want to spoof works, but that would mean that memory scanners like Moneta would be able to detect the loader very easily.

Image



Address	DLL Image	Section	Start Address	End Address
0x00007FFB56010000	R	Header	0x00000000	0x00000000
0x00007FFB56011000	RX	.text	0x00008000	0x00008000
0x00007FFB56144000	R	.rdata	0x00000000	0x00000000
0x00007FFB562C6000	RW	.data	0x00004000	0x00004000
0x00007FFB562CB000	R	.pdata	0x00001000	0x00001000
0x00007FFB562CB000	R	.didat	0x00001000	0x00001000
0x00007FFB562CB000	R	.rsrc	0x00001000	0x00001000
0x00007FFB562CB000	R	.reloc	0x00001000	0x00001000

So, we will leave this approach as a last resource.

## IAT Hooking

---

A viable alternative is IAT hooking, which means that while loading the PowerShell binary, we do not resolve the addresses of the relevant APIs correctly. Instead, we set their addresses to our own implementations that will simply mimic the real ones.

However, PowerShell does not directly import the functions that we need to hook, which means that traditional IAT hooking will not work. Nonetheless, I decided to dig deeper and try to understand how function address resolution works within PowerShell.

I selected, at random, one of the functions that use the *ConsoleHandle* (*SetConsoleMode*) and configured a software breakpoint on it. Once it got hit, I got out of the current function to inspect the function that called it.

```

0:008> bp KERNELBASE!SetConsoleMode
0:008> g
Breakpoint 1 hit
KERNELBASE!SetConsoleMode:
00007ffc`f5477640 4053          push    rbx
0:008> gu
Microsoft_PowerShell_ConsoleHost_ni+0x72d6e

```

After some inspection, I determined how this function calls *SetConsoleMode*. This is done with the following instructions:

```

00007ffc`c6e82d08 4c8955c0      mov     qword ptr [rbp-40h], r10
...
00007ffc`c6e82d44 488b4dc0      mov     rcx, qword ptr [rbp-40h]
00007ffc`c6e82d48 488b4920      mov     rcx, qword ptr [rcx+20h]
00007ffc`c6e82d4c 488b01        mov     rax, qword ptr [rcx]
...
00007ffc`c6e82d6c ffd0          call   rax <-- calls SetConsoleMode

```

The register r10 contained a pointer to some unknown structure that stores the address of *SetConsoleMode*. We can replicate this on WinDbg to find the address where the pointer of *SetConsoleMode* is stored in memory.

```

0:007> dq rbp-40h L 1
00000000`00cfe310 00007ffc`c6e236a0 <-- start of unknown struct
0:007> dq 00007ffc`c6e236a0+20h L 1
00007ffc`c6e236c0 00007ffc`c6e28490 <-- pointer stored at offset 0x20
0:007> dq 00007ffc`c6e28490 L 1
00007ffc`c6e28490 00007ffc`f68356b0 <-- address of SetConsoleMode
0:007> u 00007ffc`f68356b0
KERNEL32!SetConsoleMode:
00007ffc`f68356b0 ff2532b50500     jmp     qword ptr [KERNEL32!_imp_SetConsoleMode
(00007ffc`f6890be8)]

```

Ok, so we now know that *0x7ffc6e28490* stores the address of *SetConsoleMode*. But who sets this memory address? To find that out, I once again used a hardware breakpoint, which triggered when someone writes to that address.

After I reran everything with the hardware breakpoint set, I got a hit:

```

0:007> ba w 8 00007ffc`c6e28490
0:007> g
Breakpoint 1 hit
clr!NDirectMethodDesc::SetNDirectTarget+0x3c

```

This means that the `NDirectMethodDesc` function on the CLR (and not PowerShell) is the one who resolves the address of `SetConsoleMode`. The exact process of how the resolution works is not terribly important, so I will just explain the general idea behind it.

The CLR calls `clr!NDirect::NDirectLink`, which obtains the address of the API by calling `clr!NDirectMethodDesc::FindEntryPoint` and saves it in the aforementioned structure by calling `clr!NDirectMethodDesc::SetNDirectTarget`. The function `clr!NDirectMethodDesc::FindEntryPoint` works by calling `KERNEL32!GetProcAddressForCaller`.

We can observe how the CLR resolves the all the relevant functions in real time by setting a breakpoint on `GetProcAddressForCaller` and printing the second arguments as a string on each hit:

```
0:013> bp kernelbase!GetProcAddressForCaller "da rdx;g"
0:013> g
00007ffa`7229dff3 "GetConsoleTitle"
00000000`1ca7eb71 "GetConsoleTitleW"
00007ffa`7229e0ad "SetConsoleCtrlHandler"
00000000`03d3dbb1 "SetConsoleCtrlHandlerW"
00007ffa`75ce2af5 "GetStdHandle"
00007ffa`767c56ce "GetConsoleMode"
00007ffa`7229e003 "SetConsoleTitle"
00000000`1ca7eb81 "SetConsoleTitleW"
...
```

Now that we have a decent understanding of how resolving the address of these functions works, can we abuse it somehow? The answer is yes, because when the CLR resolves the address of `SetConsoleMode` (or any other function), it stores the pointer in a region of memory that is RW (readable and writeable), meaning we can search for these pointers and replace them with our own.

However, this approach is not straightforward given that we would need to find and modify these pointers while PowerShell is running, but only after they are resolved and before they are used. This complicates things quite a bit, so I decided to continue searching for other alternatives.

## Hardware Breakpoints

---

Using hardware breakpoints would allow us to redirect the execution of any function without patching its memory (so memory scanners will not be a concern), but the main issue with this approach is that each thread only has 4 slots for hardware breakpoints, and we need to hook more than 10 functions.

Given that the order in which the functions are called seems to be the same every time, we could simply set a hardware breakpoint in the first function and once it is called, unset it and set it in the second function and so on. This is feasible but also unnecessary, given that there is a better way.

All the functions that we need to hook, except for *CreateFile*, are just a wrapper to a lower-level API called *NtDeviceIoControlFile*. So instead of worrying about more than ten functions, we only need to worry about *NtDeviceIoControlFile* and *CreateFile*.

The second issue is that we can only set hardware breakpoints on the main thread. PowerShell will create other threads which will not have any hardware breakpoint set.

Moreover, threads created by PowerShell do indeed read the *ConsoleHandle* as can be seen next (notice the call stack does not begin in unbacked memory from Beacon):

```
# Child-SP          RetAddr           Call Site
00 KERNELBASE!GetConsoleTitleInternal+0x67 <-- function that reads the ConsoleHandle
01 KERNELBASE!GetConsoleTitleW+0x20
02 Microsoft_PowerShell_ConsoleHost_ni+0x72008
03 Microsoft_PowerShell_ConsoleHost_ni+0x5a9e8
04 Microsoft_PowerShell_ConsoleHost_ni+0x6d752
05 mscorlib_ni+0x588c87
06 mscorlib_ni+0x55fbe8
07 mscorlib_ni+0x55fad5
08 mscorlib_ni+0x589d01
09 mscorlib_ni+0x588dd1
0a mscorlib_ni+0x59ae56
0b clr!CallDescrWorkerInternal+0x83
0c clr!CallDescrWorkerWithHandler+0x47
0d clr!MethodDescCallSite::CallTargetWorker+0xfa
0e clr!QueueUserWorkItemManagedCallback+0x2a
0f clr!ManagedThreadBase_DispatchInner+0x33
10 clr!ManagedThreadBase_DispatchMiddle+0x83
11 clr!ManagedThreadBase_DispatchOuter+0x87
12 clr!ManagedThreadBase_FullTransitionWithAD+0x2f
13 clr!ManagedPerAppDomainTPCount::DispatchWorkItem+0x9a
14 clr!ThreadpoolMgr::ExecuteWorkRequest+0x51
15 clr!ThreadpoolMgr::WorkerThreadStart+0xe9
16 clr!Thread::intermediateThreadProc+0x8a
17 KERNEL32!BaseThreadInitThunk+0x14
18 ntdll!RtlUserThreadStart+0x21
```

To deal with this, we could set a hardware breakpoint on *CreateThread* so that we can configure the new thread each time that function is called. Luckily, it turns out that is not necessary because these threads do not need to work at all to recover the output. We can let them fail safely and we still get our output back.

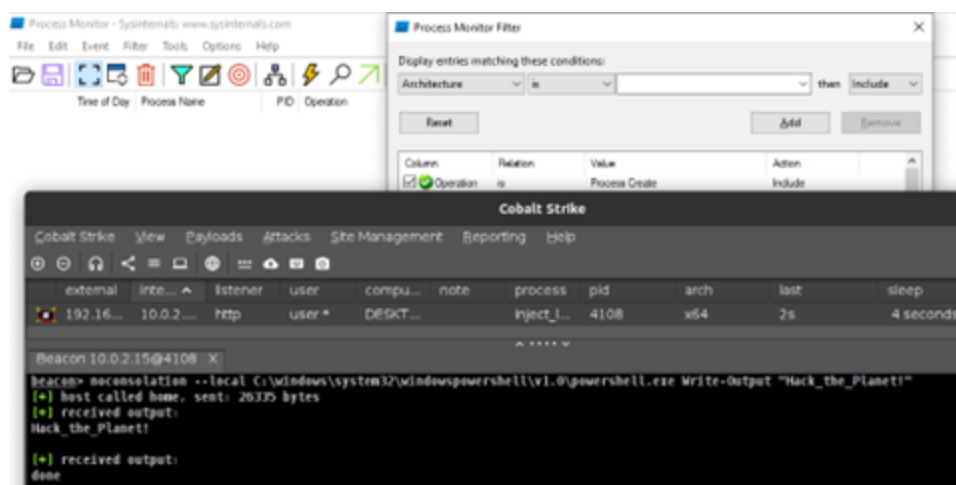
So, I only needed to set a hardware breakpoint in *NtDeviceIoControlFile* and *CreateFile* on the main thread to successfully redirect the output for PowerShell.

When I detect a call to *CreateFile*, I check the first parameter. If it is "CONOUT\$", I immediately return a value other than -1. If not, I let the execution continue.

If *NtDeviceIoControlFile* is called instead, I check the first parameter. If it is the (fake) *ConsoleHandle*, I imitate the behavior of *NtDeviceIoControlFile* when a console is present. If not, I let the execution continue.

After all this, I successfully spoofed a console on PowerShell and redirected its output.

Image



## Modifying the ConsoleHandle

Another idea that came to my mind that I wanted to share was setting the *ConsoleHandle* to a handle owned by my own loader. In theory, every time *NtDeviceIoControlFile* gets called, I would receive the message and answer it the same way the console would.

However, according to Microsoft's [documentation](#), what this function does is:

*Builds descriptors for the supplied buffer(s) and passes the untyped data to the device driver associated with the file handle.*

The description is not terribly clear, but it means that this function allows processes to interact with a device driver which can be associated with a file, a USB, or in this case a console.

In other words, *NtDeviceIoControlFile* was not designed for inter-process communication, the syscall expects a handle to a device driver and not something like a pipe or a socket, so this option is not feasible, unless someone proves me wrong 😊.

## Running PowerShell Multiple Times

---

An interesting caveat that I found during the development of this tool was that while the redirection for PowerShell worked perfectly the first time, all subsequent calls failed.

Image

```
beacon> noconsolation --local C:\windows\system32\windowspowershell\v1.0\powershell.exe $ExecutionContext.SessionState.LanguageMode
[+] host called home, sent: 26377 bytes
[+] received output:
FullLanguage

[+] received output:
done
beacon> noconsolation --local C:\windows\system32\windowspowershell\v1.0\powershell.exe $ExecutionContext.SessionState.LanguageMode
[+] host called home, sent: 26377 bytes
[+] received output:
out-Linesoutput : The handle is invalid.
+ CategoryInfo          : NotSpecified (:) [out-Linesoutput], IOException
+ FullyQualifiedErrorId : System.IO.IOException,Microsoft.PowerShell.Commands.OutLineOutputCommand

[+] received output:
done
```

This turned out to be because I was creating a new anonymous pipe on each run and closing it upon cleanup. PowerShell caches the first handle it uses for standard output and when it gets closed, the output redirection breaks down.

To counter that, I created the anonymous pipe once and reused it on all subsequent runs. To “remember” the pipe handles in between executions, I used the new key/value storage feature that Cobalt Strike launched in [release 4.9](#).

Image

```
beacon> noconsolation --local C:\windows\system32\windowspowershell\v1.0\powershell.exe $ExecutionContext.SessionState.LanguageMode
[+] host called home, sent: 26377 bytes
[+] received output:
FullLanguage

[+] received output:
done
beacon> noconsolation --local C:\windows\system32\windowspowershell\v1.0\powershell.exe $ExecutionContext.SessionState.LanguageMode
[+] host called home, sent: 26377 bytes
[+] received output:
FullLanguage

[+] received output:
done
```

Lastly, this blogpost would not be complete without a screenshot of mimikatz running:

Image



```
beacon> noconsolation /tmp/mimikatz.exe standard::coffee exit
[+] host called home, sent: 1381512 bytes
[+] received output:

.#####.   mimikatz 2.2.0 (x64) #19041 Sep 19 2022 17:44:08
.## ^ ##.   "A La Vie, A L'Amour" - (oe.eo)
## / \ ##   /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > https://blog.gentilkiwi.com/mimikatz
'## v ##'   Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'   > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(commandLine) # standard::coffee

( (
) )

┌───┐
├───┤
└───┘

mimikatz(commandLine) # exit
Bye!

[+] received output:
done
```

## Conclusion

---

This is a good example of how a deeper understanding of Windows internals can help us improve our tradecraft. While this loader is by no means undetectable, it empowers others to build upon it as I built on top of other people's work while developing it.

Each time we get mess with undocumented Windows structures and functions, we risk crashing in past and future Windows versions, given that they might change without a warning. Consider that before running this or any other tool on your Beacon, always test locally before running anything on your client's network.

Check out the tool that implements all of this [here](#).

Thank you for reading and happy hacking!